AReaL-DTE: Sparse Policy-Weight Transfer for Online Agentic Reinforcement Learning

Anonymous authors
Paper under double-blind review

Abstract

Online agentic reinforcement learning implemented with micro-services separates policy model training from rollout generation, improving scalability and modularity while potentially making frequent policy-weight synchronization a critical systems overhead. Shared storage can naturally connect these services across clusters, but vanilla dense policy-weight synchronization could be communication-intensive. Sparse synchronization reduces transferred data, yet checkpoint-oriented approaches can still retain a previous model and materialize complete intermediates to connect heterogeneous training and inference layouts. We present AReaL-DTE, a snapshot-free Delta Transfer Engine that turns inference-visible weight sparsity into end-to-end system efficiency. Across our evaluated workloads, fewer than 2% of `BF16` weight elements change between consecutive policy versions. AReaL-DTE reconstructs overwritten weights on demand by inverting AdamW updates, streams reconstructed and current parameters through converter-aligned `BF16` change detection, and remaps changed elements directly into receiver-local coordinates. AReaL-DTE supports manifest-committed sparse transfer through shared storage across clusters and a deadlock-safe two-round protocol within a cluster, followed by direct application to inference shards. We evaluate AReaL-DTE on Qwen3-8B and Qwen3-30B-A3B across four online RL workloads. AReaL-DTE achieves speedups of up to $19.9\times$ over ByteCheckpoint and $3.2\times$ over PULSE across clusters, and up to $7.6\times$ and $7.4\times$, respectively, within a cluster. In the same-cluster Qwen3-30B-A3B experiments, it reduces peak GPU memory by approximately 41% and peak CPU memory by at least 87%.

1 Introduction

Online reinforcement learning (RL), in which a policy model repeatedly generates trajectories, receives rewards, and updates its policy model, has become an important mechanism for advancing LLM capabilities for production-level deployment (Jackson et al., 2026). Supporting this RL paradigm requires a continuous loop connecting agent interaction, rollout generation, reward computation, and policy training. As these workloads grow, emerging systems increasingly deploy the stages as independently scalable components. For example, AReaL-2.0 exposes agent, inference, training, and weight-synchronization services (Yan et al., 2026); ProRL-Agent provides rollout as an independent service (Zhang et al., 2026); and RollArt (Gao et al., 2026b) and FlexMARL (Jiang et al., 2026) disaggregate rollout, environment execution, reward computation, and training across heterogeneous resources. This efficient *micro-service-oriented infrastructure* improves scalability and modularity, but such a design also raises a critical problem: *whenever training workers produce a new version of the policy model, how can the weight-synchronization service efficiently transfer its inference-usable weights to rollout inference workers?*

Consistent shared storage provides a natural rendezvous for these decoupled micro-services. During agentic RL training, the training service writes an updated weight version to shared storage, and rollout services retrieve it when synchronizing their local policy (Miahi & Belilovsky, 2026). Such a storage-mediated path allows training and rollout workers to operate with independent lifecycles, communication groups, and cluster placements (Ye et al., 2026). Without careful optimization, however, each policy update must be constructed, staged, written, transferred, read, and installed (Wan et al., 2025). Representing every update as a complete checkpoint makes both transfer volume

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

and local processing overhead grow linearly with model size, potentially dominating the interval between policy updates (Miahi & Belilovsky, 2026).

On the other hand, sparse weight synchronization offers a promising alternative. RL optimization typically changes only a small fraction of the floating-point values in the policy model visible to rollout inference workers, i.e., fewer than 2% in our evaluated workloads. Systems such as PULSE (Miahi & Belilovsky, 2026) and AURORARL (Ruan et al., 2026) exploit this property by transferring changed indices and their new values rather than the complete policy. They also demonstrate how sparse updates can be exchanged through shared storage or represented as versioned delta checkpoints. Compared with dense checkpoint mechanisms such as ByteCheckpoint (Wan et al., 2025), these approaches substantially reduce the number of bytes transferred between training and rollout inference. However, a sparse payload does not eliminate the system overhead of constructing, routing, and applying it due to the following challenges. ***First***, because AdamW updates policy weights in place, conventional change detection must retain a model-scale copy of the previous weights for calculation. ***Second***, training and rollout inference engines may fuse, permute, rename, and re-shard the same parameters differently, so a changed index in a training shard cannot be directly applied to an inference shard. Bridging these layouts through checkpoints can introduce full-state gathering, model-scale conversion, CPU-GPU staging, and resharding. ***Third***, after sparse changes are routed to their destination shards, the payload size for each sender-receiver pair is data-dependent and may be zero, complicating buffer allocation and matched communication. Efficient sparse synchronization must therefore recover overwritten weights without a historical snapshot, translate changes across heterogeneous layouts without a complete checkpoint intermediate, and safely transfer variable-length updates to their destination shards.

To overcome these challenges, we present AREAL-DTE, where DTE stands for Delta Transfer Engine, a snapshot-free sparse weight-synchronization system designed for data transfer between training and rollout inference micro-services. We make the following key contributions:

Contribution 1. We empirically identify that consecutive RL optimization steps modify only a small fraction of the weight values visible to rollout inference. Across Qwen3-8B and Qwen3-30B-A3B on GSM8K, HumanEval, MATH-500, and LogiQA, fewer than 2% of `BF16` weight elements change between consecutive policy versions. We then establish its theoretical foundation by proving a sparsity bound that connects local trajectory-KL control and directional Fisher curvature to exact `BF16` boundary-crossing geometry, thereby explaining why dense `FP32` updates can induce sparse inference-visible changes. Because rollout inference consumes `BF16` weights, elements whose `BF16` representations remain unchanged do not need to be transferred.

Contribution 2. We design and implement AREAL-DTE to reduce synchronization latency and memory without retaining historical snapshots or materializing complete-model intermediates. AREAL-DTE reconstructs previous weights through streaming AdamW inversion and detects `BF16` changes after converting both weight versions to a common space. Within a cluster, a precomputed plan maps changes to inference shards, and a deadlock-safe two-round protocol exchanges counts before indices and values. Across clusters, we publish canonical sparse updates through shared storage for application once the storage manifest is committed. Version ordering and consistency remain with the surrounding control and storage layer.

Contribution 3. We evaluate AREAL-DTE on Qwen3-8B and Qwen3-30B-A3B across four workloads. At 10 Gbps cross-cluster bandwidth, AREAL-DTE delivers 1.58–$1.92\times$ the end-to-end training throughput of PULSE and 3.40–$4.75\times$ that of BYTECHECKPOINT. AREAL-DTE also reduces synchronization latency and memory within and across clusters. On Qwen3-30B-A3B, intra-cluster peak GPU and CPU memory decrease by approximately 41% and at least 87%, respectively, relative to the lowest-memory baseline for each metric and workload. Ablations demonstrate the benefits of optimizer inversion, shared remapping, and sparse transfer.

2 KEY OBSERVATIONS AND DESIGN RATIONALE

We present four observations that make sparse policy synchronization practical, covering weight-update sparsity, weight reconstruction, layout conversion, and transfer scheduling. We then explain the sparsity theoretically and connect these observations to the design of AREAL-DTE (Figure 1).

RL communication paradigm. We enumerate our key empirical observations below:

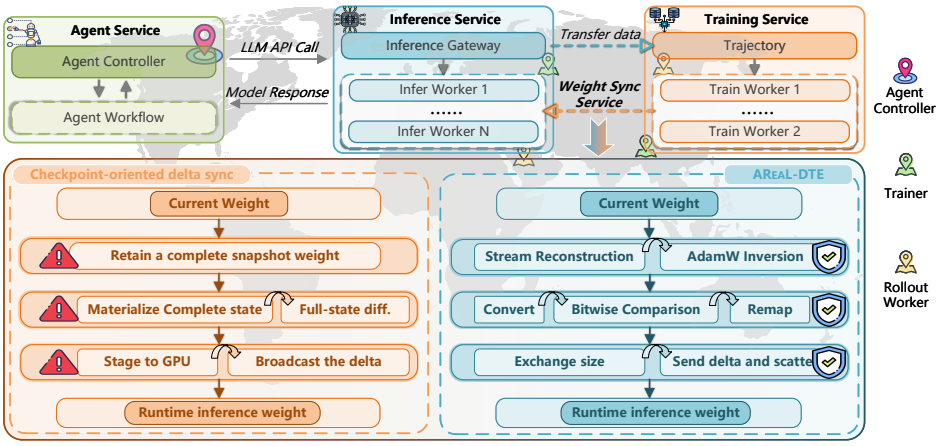


Figure 1: Motivation. Both AReaL-DTE and checkpoint-oriented delta synchronize policy deltas in online RL. Checkpoint-oriented delta synchronization may retain a historical model snapshot and materialize complete-model intermediates and stage the model between CPU and GPU. AReaL-DTE instead reconstructs previous weights on demand, detects and remaps changes during format conversion, and directly applies sparse updates to inference shards.

- *Observation 1: Policy changes are sparse in the receiver side.* Among evaluated workloads, fewer than 2% of `BF16` weight elements change between consecutive policy versions (Figure 5(c)). Updates that leave the inference-visible `BF16` weight unchanged need not be transferred.
- *Observation 2: The applied AdamW update can be inverted from the live state.* The current `FP32` master weights, moment states, optimizer step, and hyperparameters algebraically determine the pre-update weights in exact arithmetic, allowing cheap reconstruction.
- *Observation 3: Layout translation is deterministic even when layouts differ.* Training and inference engines may fuse, permute, name, and shard parameters differently, but their converter and shard metadata define a stable mapping between the two layouts. Applying the same converter to the previous and current versions creates an aligned logical space for comparison.
- *Observation 4: Transfer topology is static while payload cardinality is dynamic.* The training and inference layouts determine the sender–receiver pairs and their operation order before synchronization begins. Only the number of changed elements assigned to each pair varies across policy versions, and that number may be zero. This separation permits a fixed communication schedule even though buffer sizes are data-dependent.

Theoretical analysis. Since the communication savings essentially depend on Observation 1, we seek to explain when dense optimizer updates can produce sparse inference-visible changes. We summarize the following theorem, which connects KL control of policy updates to `BF16` rounding-boundary crossings, giving conditions under which the expected changed-weight fraction is small. This connection provides a theoretical basis for exploiting the observed sparsity during synchronization. We enumerate the main theorem below:

Theorem. *At optimizer step t, fix the pre-update policy weights $\boldsymbol{\theta}_{t-1} \in \mathbb{R}^N$ and optimizer direction $\boldsymbol{d}_t \in \mathbb{R}^N$, and consider the update $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \boldsymbol{d}_t$ with $\eta_t > 0$. Let quant_{BF16} denote rounding to `BF16`. For each coordinate i, define its critical step size as*

$$\tau_{t,i} := \inf \{a > 0 : \mathrm{quant}_{BF16}(\boldsymbol{\theta}_{t-1,i} - a\mathbf{d}_{t,i}) \neq \mathrm{quant}_{BF16}(\boldsymbol{\theta}_{t-1,i})\},$$

with $\inf \varnothing = \infty$. Define the fraction of coordinates with critical step size at most a by

$$F_t(a) := \frac{1}{N} \#\{i \in \{1, \ldots, N\} : \tau_{t,i} \leq a\}, \qquad a \geq 0,$$

and let C_t be the actual fraction of coordinates whose `BF16` values change after the update. Let $\Phi_{t,H}(\eta)$ denote the prompt-averaged KL divergence from the pre-update policy's distribution of responses of at most H tokens to the corresponding distribution after a step of size η along $-\boldsymbol{d}_t$. Assume that $\Phi_{t,H}$ is twice continuously differentiable near zero and that its directional Fisher curvature satisfies $\Phi''_{t,H}(0) \geq \chi_{t,H} > 0$. For any $\zeta \in (0, 1)$, there exists a local radius $\eta_0 > 0$ such

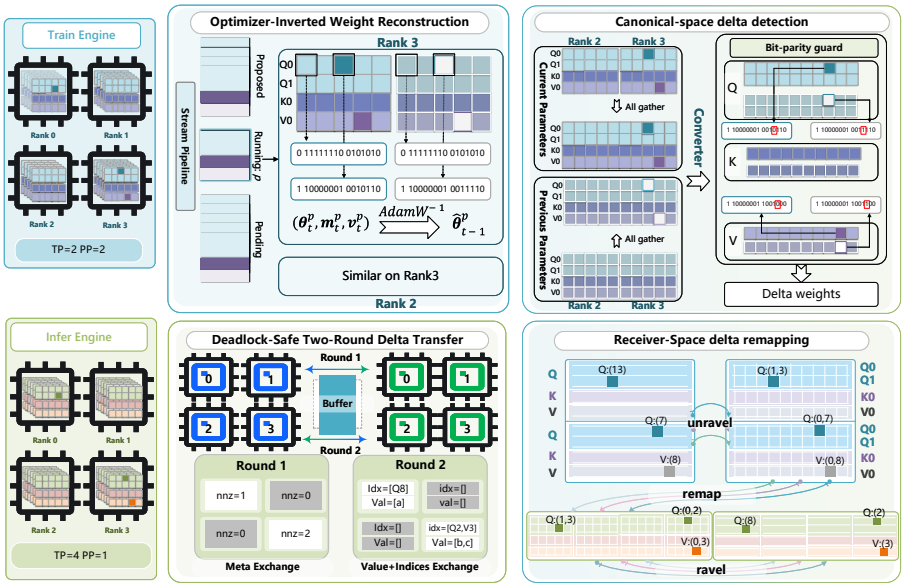


Figure 2: End-to-end same-cluster pipeline of AREAL-DTE. It reconstructs previous parameters through streaming AdamW inversion, detects inference-visible `BF16` changes after aligned conversion, maps them into receiver-local coordinates, and directly applies the resulting sparse updates to inference shards.

that every update with $0 < \eta_t \le \eta_0$ *and* $\Phi_{t,H}(\eta_t) \le \delta_{\mathrm{KL}}$*, where* $\delta_{\mathrm{KL}} > 0$*, satisfies*

$$C_t \le F_t(\eta_t) \le F_t\left(\sqrt{\frac{2\delta_{\mathrm{KL}}}{(1-\zeta)\chi_{t,H}}}\right).$$

Note that the derivation and supporting results are provided in Appendix A.

AREAL-DTE design rationale. These observations lead to a receiver-oriented design as shown in Figure 1: construct changes in the inference data type, translate them before communication, and avoid retaining or materializing state that can be derived on demand. *First*, AREAL-DTE inverts each AdamW update and streams one reconstructed converter unit at a time into change detection, eliminating the persistent historical snapshot. *Second*, we convert the reconstructed and current parameters through the same MCore-to-Hugging-Face path, compare their `BF16` bit patterns, and immediately remap changed indices into receiver-local coordinates. This fused conversion, comparison, and remapping path avoids complete-model gathering, staging, differencing, reconstruction, and resharding. *Third*, AREAL-DTE separates size discovery from data movement: we exchange a count for every planned operation, including zero-length operations, before transferring receiver-specific indices and values. Each rollout worker can then scatter the received values directly into its live inference shards without a generic-delta broadcast or receiver-side layout conversion.

3 AREAL-DTE DESIGN AND IMPLEMENTATION

Figure 2 shows how AREAL-DTE turns optimizer-native training state into sparse updates for rollout workers. It reconstructs previous weights from live AdamW state and detects inference-visible changes after training-to-inference conversion. Within a cluster, it maps these changes into receiver-local coordinates before direct communication. Across clusters, it transfers canonical sparse updates through shared storage and applies them in the destination scheduler.

3.1 OPTIMIZER-INVERTED WEIGHT RECONSTRUCTION

AdamW updates policy weights in place, so conventional change detection retains a complete pre-update policy. AREAL-DTE instead reconstructs previous parameters from the optimizer state that remains after the update, as shown in Figure 2.

Let $\boldsymbol{\theta}_t$ be an `FP32` master-weight parameter after optimizer step t, and let $\mathbf{m}_t$ and $\mathbf{v}_t$ be its first- and second-moment states. Let β_1 and β_2 denote the moment coefficients, η_t the learning rate, λ the

weight-decay coefficient, and ϵ the numerical-stability constant. In Figure 2, operator AdamW^{-1} operates element-wise on the current parameter block indexed by p, using its live state $(\boldsymbol{\theta}_t^p, \mathbf{m}_t^p, \mathbf{v}_t^p)$. For the decoupled AdamW update used by our training stack, AdamW^{-1} reconstructs the pre-update parameter as follows:

$$\widehat{\boldsymbol{\theta}}_{t-1} = \frac{\boldsymbol{\theta}_t + \eta_t \dfrac{\mathbf{m}_t/(1-\beta_1^t)}{\sqrt{\mathbf{v}_t/(1-\beta_2^t)} + \epsilon}}{1 - \eta_t \lambda}. \tag{1}$$

All quantities on the right-hand side remain available after the update, eliminating the need to save $\boldsymbol{\theta}_{t-1}$. Equation 1 is an algebraic inverse in exact arithmetic, provided that the weight-decay factor in its denominator is nonzero; Appendix C gives the derivation. In practice, optimizer rounding can prevent bitwise recovery of the previous `FP32` weights. AReaL-DTE therefore uses the result as a numerical reconstruction for `BF16` change detection.

Reconstruction takes place in the optimizer-native MCore layout, where `FP32` master weights and moment states are element-wise aligned. The running block (indexed by p) in Figure 2 is the parameter currently being reconstructed. To conduct AdamW^{-1}, each data-parallel rank reconstructs its locally owned slice, and the optimizer group gathers the slices needed to assemble the corresponding tensor-parallel shard. AReaL-DTE processes one converter unit, i.e., the parameter tensor or tensors needed for one conversion, at a time and releases reconstructed tensors after conversion and comparison. Each inverse is evaluated in parallel across tensor elements on the GPU. Temporary reconstruction state is therefore bounded by the largest converter unit rather than a full model replica.

3.2 Converter-Aligned Delta Detection and Remapping

Training and inference engines may differ in parameter names, tensor fusion, element order, data types, and sharding. An optimizer-side index therefore cannot generally be applied directly to an inference shard. AReaL-DTE detects changes in a shared canonical layout, then maps them to the appropriate inference shards.

Canonical-space delta detection. For each sender group s, AReaL-DTE passes the reconstructed parameters $\widehat{\boldsymbol{\theta}}_{t-1}$ and current parameters $\boldsymbol{\theta}_t$ through the same MCore-to-Hugging-Face converter $\mathrm{convert}_{\text{m2h}}$. The converter applies tensor-local gathering, fused-tensor splitting, permutation, and parameter renaming to both versions, producing aligned canonical tensors for comparison. Appendix D illustrates the conversion of a fused QKV parameter.

The comparison below is applied separately to each canonical tensor emitted by the current converter unit. For the running block index p, let $\mathbf{W}_{t-1,s}^p$ and $\mathbf{W}_{t,s}^p$ denote the converted reconstructed and current tensors. AReaL-DTE casts both to the `BF16` representation used by rollout inference:

$$\overline{\mathbf{W}}_{\tau,s}^p = \mathrm{quant}_{\texttt{BF16}}\left(\mathbf{W}_{\tau,s}^p\right), \qquad \tau \in \{t-1, t\}. \tag{2}$$

The bit-parity guard in Figure 2 compares the stored 16-bit patterns element-wise and extracts changed indices and their current values:

$$\mathcal{I}_{t,s}^p = \mathrm{nonzero}\left[\mathrm{vec}\left(\mathcal{B}_{16}(\overline{\mathbf{W}}_{t,s}^p) \neq \mathcal{B}_{16}(\overline{\mathbf{W}}_{t-1,s}^p)\right)\right], \quad \mathbf{V}_{t,s}^p = \mathrm{vec}\left(\overline{\mathbf{W}}_{t,s}^p\right)[\mathcal{I}_{t,s}^p]. \tag{3}$$

Here, $\mathrm{quant}_{\texttt{BF16}}(\cdot)$ casts to `BF16`, $\mathcal{B}_{16}(\cdot)$ reinterprets the stored 16-bit pattern without changing it, and $\mathrm{vec}(\cdot)$ flattens a tensor. $\mathcal{I}_{t,s}^p$ contains changed flat indices in the sender-local canonical tensor, and $\mathbf{V}_{t,s}^p$ contains the corresponding target values from $\overline{\mathbf{W}}_{t,s}^p$. This comparison excludes updates whose reconstructed and current values have identical `BF16` representations. Sending target values makes repeated application idempotent for a fixed version. Conversion and comparison operate one converter unit at a time, without materializing a complete canonical model.

Receiver-space delta remapping. For same-cluster synchronization, AReaL-DTE precomputes an ordered transfer plan $\mathcal{T}$ from the training and inference shard metadata. Each operation $\mathrm{o} \in \mathcal{T}$ identifies a block p, sender s, receiver r, and the overlap between their tensor views. Figure 2 illustrates the resulting coordinate mapping.

Let $\mathbf{n}_s^p$ and $\mathbf{n}_r^p$ be the shapes of the sender-local canonical tensor and receiver-local inference shard. Let $\mathbf{a}_s^o$ and $\mathbf{a}_r^o$ denote their overlap origins, and let $\mathbf{e}^o$ denote the overlap extent. For each

changed flat index $i \in \mathcal{I}^p_{t,s}$, AREAL-DTE first recovers its multidimensional sender coordinate $\mathbf{u} = \mathrm{unravel}(i, \mathbf{n}^p_s)$. The change belongs to operator o when $\mathbf{a}^o_s \le \mathbf{u} < \mathbf{a}^o_s + \mathbf{e}^o$ component-wise. For a change in this overlap, its receiver-local index is:

$$j_r = \mathrm{ravel}\left(\mathrm{unravel}(i, \mathbf{n}^p_s) - \mathbf{a}^o_s + \mathbf{a}^o_r, \mathbf{n}^p_r\right). \tag{4}$$

Subtracting $\mathbf{a}^o_s$ gives the coordinate within the overlap, adding $\mathbf{a}^o_r$ places it in the receiver's view, and ravel produces a receiver-local flat index. The `BF16` target value is unchanged, yielding an entry (r, j_r, v). Applying $\mathcal{T}$ partitions or replicates changes as required by the inference layout and produces an ordered payload for each sender–receiver pair. Receivers can apply these entries directly to their live shards.

3.3 DEADLOCK-SAFE TWO-ROUND DELTA TRANSFER

After remapping, each transfer-plan operation o produces receiver-local indices $\mathcal{I}_o$, target values $\mathbf{V}_o$, and an entry count $c_o = |\mathcal{I}_o|$. Sender–receiver pairs and operation order are fixed, but c_o varies across policy versions and may be zero. Receivers need these counts to allocate buffers. If one peer independently skips an empty operation, subsequent point-to-point calls can become misaligned and deadlock.

AREAL-DTE therefore uses two rounds in transfer-plan order. In the metadata round, each sender transmits c_o for every operation, and receivers allocate the corresponding buffers. In the payload round, senders transmit $\mathcal{I}_o$ and $\mathbf{V}_o$ in the same order. Both peers retain empty operations and issue matched zero-count calls, keeping subsequent operations aligned.

For a nonempty payload, receiver r directly applies the target values using

$$\mathrm{vec}(\mathbf{W}^p_r)[\mathcal{I}_o] \leftarrow \mathbf{V}_o, \tag{5}$$

where $\mathbf{W}^p_r$ denotes the live receiver-local inference shard indexed by p. After all operations for a policy version finish, AREAL-DTE notifies the control plane, which enforces version ordering and determines when the updated policy becomes visible to rollout inference.

3.4 STORAGE-MEDIATED CROSS-CLUSTER TRANSFER

Across clusters, training and inference workers cannot share a point-to-point communication group. AREAL-DTE serializes receiver-ready `BF16` index–value changes into compressed chunks and publishes them through shared storage. A version manifest is committed only after all chunks have been published. The receiving inference service verifies the delta's base version, then streams the sparse entries through the destination inference engine's weight-loading path, where the destination scheduler applies changes to live inference shards. This path avoids complete-checkpoint construction and full-model reload. Appendix E enumerates the detailed implementation.

4 EVALUATION

We evaluate AREAL-DTE along three dimensions: (*i*) end-to-end RL training efficiency (Section 4.2), (*ii*) synchronization efficiency within a cluster and across clusters at different bandwidths (Section 4.3), and (*iii*) key design choices through ablation (Section 4.4).

4.1 EXPERIMENTAL SETUP

Models and workloads. We evaluate Qwen3-8B and Qwen3-30B-A3B on four online RL workloads: GSM8K, HumanEval, MATH-500, and LogiQA.

Baselines. We compare AREAL-DTE with BYTECHECKPOINT and PULSE, which represent dense and sparse synchronization, respectively. BYTECHECKPOINT transfers a complete model state. PULSE transfers changed `BF16` values and retains the previous checkpoint for change detection. Within a cluster, we evaluate both NCCL and disk-mediated variants of each baseline. Across clusters, all methods use their storage-mediated paths.

Settings and metrics. All experiments use NVIDIA H200 GPUs, with 8 GPUs for Qwen3-8B and 16 for Qwen3-30B-A3B. Cross-cluster experiments cover effective bandwidths of 0.1, 1, and 10 Gbps. Storage-mediated artifacts are compressed with Zstandard and transferred through Alibaba Cloud Object Storage Service. We measure end-to-end training throughput from complete RL

Table 1: End-to-end RL training throughput (tokens/s) across effective cross-cluster bandwidths. Higher values are better. Note that * indicates optimal performance.

Benchmark	AReaL-DTE			ByteCheckpoint			PULSE		
	0.1 Gbps	1 Gbps	10 Gbps	0.1 Gbps	1 Gbps	10 Gbps	0.1 Gbps	1 Gbps	10 Gbps
Qwen3-8B									
GSM8K	207*	830*	1188*	11	91	306	173	552	683
HumanEval	218*	836*	1166*	12	97	328	205	563	686
MATH-500	224*	885*	1254*	13	108	369	207	623	793
LogiQA	179*	778*	1169*	12	95	327	162	520	674
Qwen3-30B-A3B									
GSM8K	46*	207*	318*	3	21	67	41	129	166
HumanEval	47*	220*	346*	3	25	77	43	140	181
MATH-500	54*	246*	383*	3	27	85	50	160	204
LogiQA	50*	230*	356*	3	25	78	47	150	196

training runs at each bandwidth. We also report synchronization latency and peak GPU and CPU memory, and compare stepwise training statistics with full-weight synchronization. Synchronization latency includes update construction, storage-mediated transfer, and receiver-side application.

4.2 End-to-End RL Training Efficiency

Training dynamics. Figure 3 compares AReaL-DTE with full-weight synchronization over 100 RL policy training steps on Qwen3-30B-A3B with GSM8K. The curves overlap perfectly throughout in gradient norm, new-policy and old-policy log probabilities, and policy entropy. This stepwise agreement shows no observable change in the measured training dynamics, supporting solid preservation of statistical efficiency.

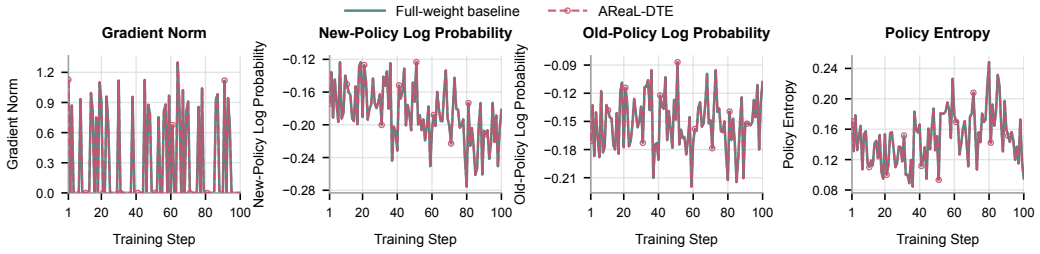


Figure 3: Training dynamics of Qwen3-30B-A3B on GSM8K over 100 steps. AReaL-DTE and full-weight synchronization overlap in all four optimization statistics.

Training throughput. Table 1 shows that AReaL-DTE achieves the highest end-to-end throughput across both models, all four workloads, and all three bandwidths. At 10 Gbps, AReaL-DTE delivers 1.58–1.92× the throughput of PULSE and 3.40–4.75× that of ByteCheckpoint. At 0.1 Gbps, network transfer dominates, narrowing the advantage over PULSE to 1.06–1.20×; the gain over ByteCheckpoint remains 14.92–18.82×. Together with the matching training dynamics, these results show that AReaL-DTE improves training throughput without an observed statistical-efficiency penalty in the tested run.

4.3 Synchronization Efficiency

Intra-cluster results. Table 2 reports synchronization latency and memory for GSM8K and HumanEval; Appendix Tables 6 and 7 cover all four workloads. These experiments measure update construction, communication, and application without a constrained inter-cluster link. Across all four workloads, AReaL-DTE synchronizes Qwen3-30B-A3B in 11.58–12.23 seconds, giving 6.8–7.6× speedup over ByteCheckpoint-NCCL and 5.1–5.5× over PULSE-NCCL. Peak GPU and CPU memory fall to 71.19–71.78 GB and 15.33–15.45 GB, respectively: approximately 41% and at least 87% below the lowest-memory baseline for each workload. On Qwen3-8B, synchronization takes

Table 2: Intra-cluster synchronization on Qwen3-30B-A3B and Qwen3-8B. Lower values are better. Note that * indicates optimal performance.

Benchmark	Metric	AReaL-DTE	ByteCheckpoint		PULSE	
			NCCL	Disk	NCCL	Disk
Qwen3-30B-A3B						
GSM8K	Latency (s)	11.58*	88.03	226.53	59.42	93.40
	GPU (GB)	71.78*	121.82	121.28	129.17	129.15
	CPU (GB)	15.33*	129.29	125.53	141.28	139.19
HumanEval	Latency (s)	11.83*	83.43	227.21	64.80	101.16
	GPU (GB)	71.45*	120.12	121.20	130.18	129.20
	CPU (GB)	15.45*	126.65	123.17	139.68	138.64
Qwen3-8B						
GSM8K	Latency (s)	2.82*	11.28	51.35	13.00	25.07
	GPU (GB)	56.05*	61.91	60.98	67.97	68.46
	CPU (GB)	3.45*	34.16	22.21	36.53	36.96
HumanEval	Latency (s)	2.86*	11.26	50.14	15.94	27.21
	GPU (GB)	57.92*	60.67	60.75	66.95	68.02
	CPU (GB)	3.45*	34.08	24.72	36.53	36.96

Table 3: Cross-cluster weight-synchronization results on Qwen3-30B-A3B. Lower values are better. Note that * indicates optimal performance.

(a) Synchronization latency (s)

Benchmark	AReaL-DTE			ByteCheckpoint			PULSE		
	0.1 Gbps	1 Gbps	10 Gbps	0.1 Gbps	1 Gbps	10 Gbps	0.1 Gbps	1 Gbps	10 Gbps
GSM8K	477.47*	71.31*	30.70*	8741.18	1080.12	314.05	544.86	140.90	99.43
HumanEval	539.64*	79.88*	33.91*	8741.01	1080.69	314.73	604.60	152.01	107.34
MATH-500	523.59*	77.88*	33.32*	8740.88	1081.11	315.14	567.42	144.20	102.75
LogiQA	511.81*	76.45*	32.92*	8744.39	1082.10	315.85	559.26	141.93	97.78

(b) Peak memory consumption (GB)

Benchmark	AReaL-DTE		ByteCheckpoint		PULSE	
	GPU	CPU	GPU	CPU	GPU	CPU
GSM8K	71.52*	40.56*	121.28	125.53	129.15	139.19
HumanEval	71.96*	37.58*	121.20	123.17	129.20	138.64
MATH-500	71.84*	37.44*	121.55	122.79	129.10	139.83
LogiQA	71.88*	40.75*	121.87	121.49	128.29	140.50

2.77–2.87 seconds, with speedups of up to $4.0\times$ and $7.4\times$ over the respective NCCL baselines. Peak CPU memory is only 3.45–3.48 GB. The results show the benefit of removing historical snapshots and complete-model intermediates even without a constrained network link.

Cross-cluster results. Table 3 reports synchronization latency at 0.1, 1, and 10 Gbps and peak memory for Qwen3-30B-A3B; Appendix Table 8 gives the Qwen3-8B results. AReaL-DTE has the lowest latency in every setting. Speedups reach $18.3\times$ over ByteCheckpoint and $3.2\times$ over PULSE on Qwen3-30B-A3B, and $19.9\times$ and $2.5\times$, respectively, on Qwen3-8B. At 0.1 Gbps, network transfer dominates both sparse methods. As bandwidth increases, local construction and application account for more of the latency, widening AReaL-DTE's advantage over PULSE. For example, on Qwen3-30B-A3B with GSM8K, this speedup grows from $1.14\times$ at 0.1 Gbps to $3.24\times$ at 10 Gbps. AReaL-DTE also reduces peak GPU and CPU memory by at least 40% and 66%, respectively, on Qwen3-30B-A3B relative to the lowest-memory storage-mediated baseline for each workload. The corresponding minimum reductions on Qwen3-8B are 13.7% and 16.0%.

4.4 Ablation Study

We evaluate three ablated variants on both models and all four workloads using the same storage-mediated transport. Appendix Table 9 reports the full results. The complete AReaL-DTE design achieves the lowest latency and peak GPU and CPU memory in every setting.

Optimizer inversion. "w/o inversion" retains the pre-update policy in host memory and stages it to the GPU for change detection. Compared with this variant, AReaL-DTE is up to $2.30\times$ faster on Qwen3-8B and $1.60\times$ faster on Qwen3-30B-A3B, while reducing peak GPU memory by up to 51.5% and 46.3%, respectively. Reconstructing weights on demand avoids both snapshot storage and staging costs.

Receiver-space remapping. "w/o inversion and remapping" additionally moves mapping and filtering to each inference rank. Comparing it with "w/o inversion" isolates the effect of remapping: removing this stage increases latency by 16.6–31.2% on Qwen3-8B and 20.3–23.6% on Qwen3-30B-A3B. Sharing the remapping work avoids repeating it on every inference rank.

Sparse transfer. "w/o sparse transfer" synchronizes the complete policy. Compared with this variant, AReaL-DTE is up to $7.9\times$ faster on Qwen3-8B and $4.5\times$ faster on Qwen3-30B-A3B. Thus, sparse payloads and reduced local processing both contribute to synchronization efficiency.

5 Related Work

RL post-training systems. Distributed RL frameworks coordinate rollout generation, reward computation, and policy optimization across specialized workers (Sheng et al., 2025b; Hu et al., 2024; Shen et al., 2024; Wang et al., 2025). Recent systems further disaggregate these stages or introduce asynchronous rollout, policy versioning, relay workers, and cross-cluster scheduling (Fu et al., 2025; Yan et al., 2026; Zhang et al., 2026; Gao et al., 2026b; Jiang et al., 2026; Hu et al., 2026; Sheng et al., 2025a; Wu et al., 2026; Gao et al., 2026a). While much of this work focuses on execution schedules and resource allocation, AReaL-DTE targets the data path used to transfer each updated policy from training to rollout workers.

Model-state transfer across heterogeneous computation platforms. Training frameworks such as Megatron-LM (Narayanan et al., 2021) and FSDP (Zhao et al., 2023) and inference engines such as vLLM (Kwon et al., 2023) and SGLang (Zheng et al., 2024) may represent the same policy using different naming, fusion, permutation, data types, and parallel layouts. ByteCheckpoint (Wan et al., 2025) and Universal checkpointing (Lian et al., 2025) provide portable full-state representations, while TensorHub (Ye et al., 2026) serves topology-aware transfers from live replicas. These mechanisms address portability or transfer topology, but each refresh still processes a complete model. AReaL-DTE instead preserves layout translation without materializing a complete intermediate policy.

Sparse policy-weight synchronization. PULSE (Miahi & Belilovsky, 2026) transmits only the indices and new `BF16` values of weights whose `BF16` representations change. AuroraRL (Ruan et al., 2026), ROSE (Gao et al., 2026a), and Slime (slime Contributors, 2026) also reduce communication by packaging or routing sparse policy deltas to rollout workers. These approaches reduce communication volume, but delta construction or application may still require a previous-model snapshot, canonical checkpoint, full-state conversion, or receiver-side reconstruction. AReaL-DTE makes the local pipeline sparse as well: it reconstructs overwritten weights on demand, detects changes during training-to-inference conversion, and routes them for direct application without a complete-model intermediate.

6 Conclusion

We present AReaL-DTE, a sparse policy-weight synchronization system for online agentic RL. AReaL-DTE combines streaming AdamW inversion, converter-aligned `BF16` change detection, and layout-aware sparse transfer to avoid persistent historical snapshots and complete-model intermediates. Across two Qwen3 models and four workloads, AReaL-DTE reduces synchronization latency and memory consumption within and across clusters. At 10 Gbps cross-cluster bandwidth, it delivers 1.58–$1.92\times$ the end-to-end training throughput of PULSE and 3.40–$4.75\times$ that of ByteCheckpoint. These results demonstrate that coordinating sparse update construction, transfer, and application translates inference-visible weight sparsity into practical performance gains.

AI USE STATEMENT

The authors used large language model-based tools to aid and polish writing, draft and restructure portions of the manuscript, and assist with LaTeX formatting. These tools were also used to support research ideation and execution, including brainstorming analysis directions, organizing and presenting experimental analyses, preparing visualizations, and checking statistics. All AI-assisted outputs were reviewed and verified by the authors, who made the final decisions regarding the technical content, experiments, analyses, and presentation. The authors take full responsibility for the accuracy and content of the paper.

REPRODUCIBILITY STATEMENT

The implementation of AREAL-DTE is available in an anonymous repository at `https://anonymous.4open.science/r/q7m4x2k9/`.

The datasets used in our evaluation are publicly available at:

- GSM8K: `https://github.com/openai/grade-school-math`.
- HumanEval: `https://github.com/openai/human-eval`.
- MATH-500: `https://huggingface.co/datasets/HuggingFaceH4/MATH-500`.
- LogiQA: `https://github.com/lgw863/LogiQA-dataset`.

REFERENCES

Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, Jiashu Wang, Tongkai Yang, Binhang Yuan, and Yi Wu. AReaL: A large-scale asynchronous reinforcement learning system for language reasoning. In *Advances in Neural Information Processing Systems*, volume 38, pp. 36256–36282. Curran Associates, Inc., 2025. doi: 10.52202/085713-1218. URL `https://proceedings.neurips.cc/paper_files/paper/2025/hash/33c00862bfa29ac72ecf630a41e19352-Abstract-Conference.html`.

Wei Gao, Yuheng Zhao, Dilxat Muhtar, Dakai An, Xuchun Shang, Tianyuan Wu, Lunxi Cao, Shaopan Xiong, Weixun Wang, Ju Huang, Teng Ma, Siran Yang, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. ROSE: Rollout on serving GPUs via cooperative elasticity for agentic RL. *arXiv preprint arXiv:2605.06534*, 2026a. doi: 10.48550/arXiv.2605.06534. URL `https://arxiv.org/abs/2605.06534`.

Wei Gao, Yuheng Zhao, Tianyuan Wu, Shaopan Xiong, Weixun Wang, Dakai An, Lunxi Cao, Dilxat Muhtar, Zichen Liu, Haizhou Zhao, Ju Huang, Siran Yang, Yongbin Li, Wenbo Su, Jiamang Wang, Lin Qu, Bo Zheng, and Wei Wang. RollArt: Disaggregated Multi-Task agentic RL training at scale. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 863–881. USENIX Association, July 2026b. URL `https://www.usenix.org/conference/osdi26/presentation/gao`.

Jian Hu, Xibin Wu, Zilin Zhu, Xianyu, Weixun Wang, Dehao Zhang, and Yu Cao. OpenRLHF: An easy-to-use, scalable and high-performance RLHF framework. *arXiv preprint arXiv:2405.11143*, 2024. URL `https://arxiv.org/abs/2405.11143v4`.

Tianhao Hu, Xiangcheng Liu, Yuchun Miao, Youshao Xiao, Hongyu Zang, Yang Zheng, Xuan Huang, Jinrui Ding, Yufei Zhang, Yu Yang, Yi-Kai Zhang, Yueqing Sun, Chengcheng Han, Xiandi Ma, Wei Wang, Qi Gu, Yerui Sun, Yuchen Xie, and Xunliang Cai. DORA: A scalable asynchronous reinforcement learning system for language model training. *arXiv preprint arXiv:2604.26256*, 2026. doi: 10.48550/arXiv.2604.26256. URL `https://arxiv.org/abs/2604.26256`.

Jacob Jackson, Ben Trapani, Nathan Wang, and Wanqi Zhu. Improving Composer through real-time RL. `https://cursor.com/blog/real-time-rl-for-composer`, March 2026. Accessed: 2026-07-01.

Zhida Jiang, Zhaolong Xing, Jiawei Lu, Yipei Niu, Qingyuan Sang, Liangxu Zhang, Wenquan Dai, Junhua Shu, Jiaxing Wang, Qiangyu Pei, Qiong Chen, Xinyu Liu, Fangming Liu, Ai Han, Zhen

Chen, and Ke Zhang. Rollout-training co-design for efficient LLM-based multi-agent reinforcement learning. *arXiv preprint arXiv:2602.09578*, 2026.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. doi: 10.1145/3600006.3613165. URL https://arxiv.org/abs/2309.06180.

Xinyu Lian, Sam Ade Jacobs, Lev Kurilenko, Masahiro Tanaka, Stas Bekman, Olatunji Ruwase, and Minjia Zhang. Universal checkpointing: A flexible and efficient distributed checkpointing system for large-scale DNN training with reconfigurable parallelism. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*, pp. 1519–1534. USENIX Association, July 2025. URL https://www.usenix.org/conference/atc25/presentation/lian.

Erfan Miahi and Eugene Belilovsky. Understanding and exploiting weight update sparsity for communication-efficient distributed rl. *arXiv preprint arXiv:2602.03839*, 2026.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021. doi: 10.1145/3458817.3476209. URL https://arxiv.org/abs/2104.04473.

Chaoyi Ruan, Geng Luo, Xinyi Wan, Long Zhao, Qinghe Wang, Jiaan Zhu, Duling Xu, Guanbin Xu, Dehui Wei, Xiang Liu, Cheng Li, Haifeng Sun, Liang Luo, Congcong Miao, and Jialin Li. AuroraRL: Fast, fault-tolerant, and cost-efficient reinforcement learning over decentralized network. *arXiv preprint arXiv:2602.11456*, 2026. doi: 10.48550/arXiv.2602.11456. URL https://arxiv.org/abs/2602.11456.

Gerald Shen, Zhilin Wang, Olivier Delalleau, Jiaqi Zeng, Yi Dong, Daniel Egert, Shengyang Sun, Jimmy J. Zhang, Sahil Jain, Ali Taghibakhshi, Markel Sanz Ausin, Ashwath Aithal, and Oleksii Kuchaiev. NeMo-Aligner: Scalable toolkit for efficient model alignment. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=yK2eGE8QVW.

Guangming Sheng, Yuxuan Tong, Borui Wan, Wang Zhang, Chaobo Jia, Xibin Wu, Yuqi Wu, Xiang Li, Chi Zhang, Yanghua Peng, Haibin Lin, Xin Liu, and Chuan Wu. Laminar: A scalable asynchronous RL post-training framework. *arXiv preprint arXiv:2510.12633*, 2025a. doi: 10.48550/arXiv.2510.12633. URL https://arxiv.org/abs/2510.12633.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. HybridFlow: A flexible and efficient RLHF framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297. ACM, 2025b. doi: 10.1145/3689031.3696075.

slime Contributors. slime documentation: Delta weight sync. https://github.com/THUDM/slime/blob/main/docs/en/advanced/delta-weight-sync.md, 2026. Accessed: 2026-07-01.

Borui Wan, Mingji Han, Yiyao Sheng, Yanghua Peng, Haibin Lin, Mofan Zhang, Zhichao Lai, Menghan Yu, Junda Zhang, Zuquan Song, Xin Liu, and Chuan Wu. ByteCheckpoint: A unified checkpointing system for large foundation model development. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pp. 559–578. USENIX Association, April 2025. URL https://www.usenix.org/conference/nsdi25/presentation/wan-borui.

Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, Zichen Liu, Haizhou Zhao, Dakai An, Lunxi Cao, Qiyang Cao, Wanxi Deng, Feilei Du, Yiliang Gu, Jiahe Li, Xiang Li, Mingjie Liu, Yijia Luo, Zihe Liu, Yadao Wang, Pei Wang, Tianyuan Wu, Yanan Wu, Yuheng Zhao, Shuaibing Zhao, Jin Yang, Siran Yang, Yingshui Tan, Huimin Yi, Yuchi Xu, Yujin Yuan, Xingyao Zhang, Lin Qu, Wenbo

Su, Wei Wang, Jiamang Wang, and Bo Zheng. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library. *arXiv preprint arXiv:2506.06122*, 2025.

Tianyuan Wu, Lunxi Cao, Yining Wei, Wei Gao, Yuheng Zhao, Dakai An, Shaopan Xiong, Zhiqiang Lv, Ju Huang, Siran Yang, Yinghao Yu, Jiamang Wang, Lin Qu, and Wei Wang. Weave: Efficient co-scheduling for disaggregated RL post-training. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*, pp. 809–827. USENIX Association, July 2026. URL https://www.usenix.org/conference/osdi26/presentation/wu-tianyuan.

Ran Yan, Wei Fu, Jiale Li, Shusheng Xu, Zhiyu Mei, Jiaxuan Gao, Jiarui Zhang, Wentai Zhang, Hao Dai, Xujie Shen, Chuyi He, Zhen Pu, Jun Mei, Zhiyao Lin, Haitao Wang, Zhiqiang Ding, Jiawei Zhang, Huaijie Wang, Ruida Xu, Honghua Dong, Youhe Jiang, Yi Wu, Tongkai Yang, and Binhang Yuan. Next-generation agentic reinforcement learning systems enable self-evolving agents. *arXiv preprint arXiv:2607.01120*, 2026. doi: 10.48550/arXiv.2607.01120. URL https://arxiv.org/abs/2607.01120.

Chenhao Ye, Huaizheng Zhang, Mingcong Han, Baoquan Zhong, Xiang Li, Qixiang Chen, Xinyi Zhang, Weidong Zhang, Kaihua Jiang, Wang Zhang, He Sun, Wencong Xiao, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. TensorHub: Scalable and elastic weight transfer for LLM RL training. *arXiv preprint arXiv:2604.09107*, 2026. doi: 10.48550/arXiv.2604.09107. URL https://arxiv.org/abs/2604.09107.

Hao Zhang, Mingjie Liu, Shaokun Zhang, Songyang Han, Jian Hu, Zhenghui Jin, Yuchi Zhang, Shizhe Diao, Ximing Lu, Binfeng Xu, Zhiding Yu, Jan Kautz, and Yi Dong. ProRL Agent: Rollout-as-a-service for RL training of multi-turn LLM agents. *arXiv preprint arXiv:2603.18815*, 2026.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16 (12):3848–3860, 2023. doi: 10.14778/3611540.3611569. URL https://www.vldb.org/pvldb/vol16/p3848-huang.pdf.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems*, volume 37, pp. 62557–62583. Curran Associates, Inc., 2024. doi: 10.52202/079017-2000. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/724be4472168f31ba1c9ac630f15dec8-Abstract-Conference.html.

A A Mathematical Account of Inference-Visible Weight Sparsity

This section explains why a dense FP32 optimizer update can induce a sparse change in the BF16 policy used for rollout inference. The argument has two logically distinct parts. First, in reinforcement learning, policy updates are constrained by a KL-divergence budget, which requires sufficiently small learning rates. Second, when the learning rate is sufficiently small, only a small fraction of coordinates cross a BF16 rounding boundary.

Specifically, consider a parameterized policy model $\pi_{\boldsymbol{\theta}}$ with parameter $\boldsymbol{\theta}$, and let the optimizer update be

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta_t \boldsymbol{d}_t, \quad \eta_t > 0, \tag{6}$$

Here, t denotes the iteration step, $\boldsymbol{\theta}_t$ denotes the updated model parameters after iteration t, η_t is the learning rate at step t, and $\boldsymbol{d}_t$ is the realized preconditioned optimizer direction. For autoregressive generation tasks, we assume a fixed response horizon H, with the EOS token allowing early

stopping. The token-level score function can be defined as

$$\boldsymbol{s}_\ell := A_\ell \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}_{t-1}}(Y_\ell \mid h_\ell), \tag{7}$$

where $A_\ell = 1$ if ℓ is at or before the first EOS token and 0 otherwise. The history before token ℓ is $h_\ell = (x, Y_{<\ell})$, where x is the input and Y_ℓ is the ℓ-th token in the response.

We use the following function of the KL divergence between the pre-update and post-update models to constrain the new model from deviating too far from the old one:

$$\Phi_{t,H}(\eta) := \mathbb{E}_x \left[D_{\mathrm{KL}} \left(P^{(H)}_{\boldsymbol{\theta}_{t-1}}(\cdot \mid x) \| P^{(H)}_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(\cdot \mid x) \right) \right], \tag{8}$$

Here, $P^{(H)}_{\boldsymbol{\theta}}$ denotes the distribution of sequences of horizon H generated by policy $\pi_{\boldsymbol{\theta}}$; tokens after EOS are treated as undifferentiated, namely,

$$P^{(H)}_{\boldsymbol{\theta}}(Y_{1:H} \mid x) = \prod_{\ell=1}^{H} \tilde{\pi}_{\boldsymbol{\theta}}(Y_\ell \mid h_\ell), \tag{9}$$

where $\tilde{\pi}_{\boldsymbol{\theta}}$ uses the policy distribution when the token is active and otherwise uses the fixed absorbing distribution δ_{absorb}, which is independent of $\boldsymbol{\theta}$. The following theorem gives an upper bound on the update step size η_t under a fixed KL-divergence budget.

Theorem 1. Assume that $\Phi_{t,H}$ is twice continuously differentiable in a neighborhood of zero. For any $\zeta \in (0, 1)$, KL-divergence budget $\delta_{\mathrm{KL}} > 0$, and sufficiently small step size $\eta > 0$, if $\Phi_{t,H}(\eta_t) \leq \delta_{\mathrm{KL}}$, then

$$\eta_t \leq \sqrt{\frac{2\delta_{\mathrm{KL}}}{(1-\zeta)\chi_{t,H}}}, \tag{10}$$

where $\chi_{t,H}$ is defined by

$$\chi_{t,H} := \sup_{\tau > 0} \tau \sum_{\ell=1}^{H} \mathbb{P}\left(|\tilde{\boldsymbol{s}}'_{\ell,\boldsymbol{d}_t}(0)| \geq \sqrt{\tau} \right), \tag{11}$$

and $\tilde{\boldsymbol{s}}_{\ell,\boldsymbol{d}_t}(\eta) := A_\ell \log \pi_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(Y_\ell \mid h_\ell)$.

Proof. By definition, $\Phi_{t,H}(0) = 0$. Differentiating once gives

$$\Phi'_{t,H}(\eta) = -\mathbb{E}_x \left[\sum_{\ell=1}^{H} \mathbb{E}_{Y_{1:\ell-1}} \left[A_\ell \sum_{Y_\ell} \pi_{\boldsymbol{\theta}_{t-1}}(Y_\ell \mid h_\ell) \frac{\partial \pi_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(Y_\ell \mid h_\ell)/\partial \eta}{\pi_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(Y_\ell \mid h_\ell)} \Big| x \right] \right]. \tag{12}$$

Because $\sum_{Y_\ell} \pi_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(Y_\ell \mid h_\ell) = 1$, it follows that $\Phi'_{t,H}(0) = 0$. Differentiating once more yields

$$\Phi''_{t,H}(\eta) = -\mathbb{E}_x \left[\sum_{\ell=1}^{H} \mathbb{E}_{Y_{1:\ell-1}} \left[A_\ell \sum_{Y_\ell} \pi_{\boldsymbol{\theta}_{t-1}}(Y_\ell \mid h_\ell) \left(\frac{\partial^2 \pi_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(Y_\ell \mid h_\ell)/(\partial \eta)^2}{\pi_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(Y_\ell \mid h_\ell)} \right. \right. \right. \tag{13}$$

$$\left. \left. \left. - \frac{(\partial \pi_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(Y_\ell \mid h_\ell)/\partial \eta)^2}{(\pi_{\boldsymbol{\theta}_{t-1} - \eta \boldsymbol{d}_t}(Y_\ell \mid h_\ell))^2} \right) \Big| x \right] \right], \tag{14}$$

Therefore,

$$\Phi''_{t,H}(0) = \mathbb{E}_x \left[\sum_{\ell=1}^{H} \mathbb{E}_{Y_{1:\ell-1}} \left[A_\ell \sum_{Y_\ell} \frac{(\nabla \pi_{\boldsymbol{\theta}_{t-1}}(Y_\ell \mid h_\ell)^\top \boldsymbol{d}_t)^2}{\pi_{\boldsymbol{\theta}_{t-1}}(Y_\ell \mid h_\ell)} \Big| x \right] \right] \tag{15}$$

$$= \mathbb{E}_x \left[\sum_{\ell=1}^{H} \mathbb{E}_{Y_{1:\ell-1}} \left[A_\ell \mathbb{E}_{Y_\ell} \left[(\nabla \log \pi_{\boldsymbol{\theta}_{t-1}}(Y_\ell \mid h_\ell)^\top \boldsymbol{d}_t)^2 \mid h_\ell \right] \mid x \right] \right] \tag{16}$$

$$= \sum_{\ell=1}^{H} \mathbb{E}[(\tilde{\boldsymbol{s}}'_{\ell,\boldsymbol{d}_t}(0))^2] \geq \chi_{t,H} \tag{17}$$

By twice continuous differentiability, $\Phi''_{t,H}(\eta) \geq (1-\zeta)\chi_{t,H}$ for sufficiently small η. Applying Taylor's formula with integral remainder gives

$$\Phi_{t,H}(\eta) = \Phi_{t,H}(0) + \eta\Phi'_{t,H}(0) + \int_0^\eta (\eta - u)\Phi''_{t,H}(u)\mathrm{d}u \geq \frac{(1-\zeta)\chi_{t,H}}{2}\eta^2, \tag{18}$$

which proves Equation 10. □

We next explain how small learning rates lead to sparse `BF16` updates. Let $\psi_{t,i}(u) = \theta_{t-1,i} - u\mathrm{sgn}(d_{t,i})$ denote the value of coordinate i after moving a distance $u > 0$ along the direction $-\boldsymbol{d}_t$. Define the critical learning rate as the smallest learning rate that produces a different `BF16` representation:

$$D^{\rightarrow}_{t,i} := \inf\{u > 0 : \mathrm{quant}_{\texttt{BF16}}(\psi_{t,i}(u)) \neq \mathrm{quant}_{\texttt{BF16}}(\psi_{t,i}(0))\}, \quad \text{with convention } \inf\varnothing = \infty, \tag{19}$$

$$\eta^\star_{t,i} := \frac{D^{\rightarrow}_{t,i}}{|d_{t,i}|}, \quad \text{if } d_{t,i} \neq 0; \quad \text{else } \eta^\star_{t,i} := \infty. \tag{20}$$

Accordingly, the following visibility indicator records whether the `BF16` update of coordinate i is visible:

$$X_{t,i}(\eta) := \mathbf{1}\{\mathrm{quant}_{\texttt{BF16}}(\psi_{t,i}(\eta|d_{t,i}|)) \neq \mathrm{quant}_{\texttt{BF16}}(\psi_{t,i}(0))\} = \mathbf{1}\{\eta^\star_{t,i} < \eta\} + T^{\mathrm{tie}}_{t,i}\mathbf{1}\{\eta^\star_{t,i} = \eta\}, \tag{21}$$

where $T^{\mathrm{tie}}_{t,i} := X_{t,i}(\eta^\star_{t,i})$. The following theorem characterizes the changed-coordinate fraction for a small learning rate.

Theorem 2. For a finite set of N coordinates, with weights $\boldsymbol{\theta}_{t-1}$ and direction $\boldsymbol{d}_t$ fixed, the changed-`BF16` fraction is

$$C_t(\eta) := \frac{1}{N}\sum_{i=1}^N \mathbb{E}[X_{t,i}(\eta)] = F_{\eta^\star_t}(\eta), \tag{22}$$

where $F_{\eta^\star_t}$ is the empirical CDF of the critical learning rates across coordinates. We use the continuous-distribution convention $\mathbb{P}(\eta^\star_{t,i} = \eta) = 0$ to ignore ties.

Proof. By definition,

$$C_t(\eta) = \frac{1}{N}\sum_{i=1}^N \mathbb{P}(\eta^\star_{t,i} \leq \eta) = \int_0^\eta \frac{1}{N}\sum_{i=1}^N p_{\eta^\star_{t,i}}(x)\mathrm{d}x, \tag{23}$$

where $p_{\eta^\star_{t,i}}$ is the PDF of $\eta^\star_{t,i}$. The empirical PDF across coordinates is $p_{\eta^\star_t} = \frac{1}{N}\sum_{i=1}^N p_{\eta^\star_{t,i}}$; hence,

$$C_t(\eta) = \int_0^\eta p_{\eta^\star_t}(x)\mathrm{d}x = F_{\eta^\star_t}(\eta), \tag{24}$$

which concludes the proof. □

Remark. Theorem 2 implies that, as $\eta \to 0$, the fraction of visible `BF16` changes also tends to zero. If, in addition, the relative position of each coordinate $\theta_{t,i}$ within its local `BF16` cell is uniformly distributed, namely, $D^{\rightarrow}_{t,i}/h_{t,i} \sim \mathrm{Uniform}[0,1]$, where $h_{t,i} := \sup\{x|\mathcal{Q}_{\texttt{BF16}}(x) = \mathcal{Q}_{\texttt{BF16}}(\theta_{t-1,i})\} - \inf\{x|\mathcal{Q}_{\texttt{BF16}}(x) = \mathcal{Q}_{\texttt{BF16}}(\theta_{t-1,i})\}$ is the length of the local `BF16` rounding cell, we have

$$C_t(\eta) = \frac{1}{N}\sum_{i=1}^N \mathbb{P}\left(\frac{D^{\rightarrow}_{t,i}}{h_{t,i}} \leq \frac{|d_{t,i}|}{h_{t,i}}\eta\right) \leq \eta\mathbb{E}[Z_t], \tag{25}$$

where $Z_t := \frac{1}{N}\sum_{i=1}^N \frac{|d_{t,i}|}{h_{t,i}}$. Combining this inequality with Theorem 1, for any KL-divergence budget $\delta_{\mathrm{KL}} > 0$ and sufficiently small learning rates, the expected fraction of visible `BF16` updates is bounded by

$$C_t \leq \mathbb{E}[Z_t]\sqrt{\frac{2\delta_{\mathrm{KL}}}{(1-\zeta)\chi_{t,H}}} = \mathcal{O}\left(\sqrt{\frac{\delta_{\mathrm{KL}}}{\chi_{t,H}}}\right). \tag{26}$$

Table 4: 95% lower confidence bounds for $\chi_{t,H}/H$ estimated during Qwen3-0.6B training. Each entry reports the confidence bound and the quantile attaining the maximum among the eight pilot choices. The highlighted entry is the weakest observed case.

Step t	$H = 64$	$H = 128$	$H = 256$	$H = 512$
13	1.3677×10^{5} $(q_{0.80})$	8.5522×10^{4} $(q_{0.80})$	1.0924×10^{4} $(q_{0.70})$	69.2399 $(q_{0.60})$
17	1.3143×10^{5} $(q_{0.80})$	6.9035×10^{4} $(q_{0.80})$	9.8354×10^{3} $(q_{0.70})$	25.9628 $(q_{0.50})$
20	2.5220×10^{5} $(q_{0.85})$	1.2529×10^{5} $(q_{0.80})$	1.5874×10^{4} $(q_{0.70})$	36.3766 $(q_{0.50})$

Table 5: Counts and rates of `FP32` and `BF16` changes in the Qwen3-30B-A3B steps 11-20 window. Each observation corresponds to one parameter element at one optimizer step; the denominator therefore contains ten times the per-step valid coverage.

Event and denominator	**Integer count ratio**	**Rate**
FP32 master changed / valid	275,636,979,353/305,321,226,240	90.2777%
BF16-visible changed / valid	5,730,973,890/305,321,226,240	1.8770%
BF16-visible / FP32 changed	5,730,973,890/275,636,979,353	2.0792%
FP32-only / FP32 changed	269,906,005,463/275,636,979,353	97.9208%
$r_{\mathrm{upd}} < 0.1$ / FP32 changed	264,566,603,247/275,636,979,353	95.9837%
$r_{\mathrm{upd}} < 0.01$ / FP32 changed	177,216,662,926/275,636,979,353	64.2935%

Note. $r_{\mathrm{upd}} := |\Delta|/U^{\mathrm{dir}}_{\mathrm{BF16}}$, where $U^{\mathrm{dir}}_{\mathrm{BF16}}$ is the local spacing between adjacent BF16 grid points along the realized update direction.

As a complementary empirical check, we estimate lower bounds on $\chi_{t,H}/H$. Specifically, we train Qwen3-0.6B on GSM8K and estimate $\max_{\tau\in\{\tau_i\}} \frac{\tau}{H}\sum_{\ell=1}^{H}\mathbb{P}(|\tilde{\boldsymbol{s}}'_{\ell,\boldsymbol{d}_t}(0)| \geq \sqrt{\tau}) - \varepsilon$ as a simultaneous 95% lower confidence bound for $\chi_{t,H}/H$. The bound covers all combinations of steps $t \in \{13, 17, 20\}$, horizons $H \in \{64, 128, 256, 512\}$, and eight candidate values of τ. These candidates are selected as the empirical quantiles $\{0.50, 0.60, 0.70, 0.80, 0.85, 0.90, 0.95, 0.99\}$ of 32 pilot trajectories, obtained from two trials on each of 16 sampled GSM8K tasks. The reported values use 256 trajectories, obtained from two trials on each of 128 sampled GSM8K tasks. By Hoeffding's inequality, $\varepsilon = \sqrt{\frac{\log(96/0.05)}{2\times128}} \approx 0.1718$. As shown in Table 4, the estimated lower bound on $\chi_{t,H}/H$ is at least 25.9628. This positive certificate is consistent with sparse `BF16`-update visibility under a small KL-divergence budget.

In addition, we examine whether the sparsity of `BF16` updates is mainly due to update magnitudes that are too small to cross local `BF16` boundaries. For the complete Qwen3-30B-A3B run, we aggregate the steps 11-20 window. As shown in Table 5, 90.2777% of valid coordinates change in the `FP32` master weights, whereas only 1.8770% are visible after conversion to `BF16`. Moreover, among parameter updates that change the FP32 master weights, 95.9837% have magnitudes below 0.1 times the corresponding minimum magnitude required to produce a visible BF16 change, while 64.2935% are below 0.01 times this threshold. Figure 4 also illustrates the step-wise visibility rate of both `FP32` and `BF16` changes.

B Additional Experimental Results

B.1 Complete Same-Cluster Results

Tables 6 and 7 report the complete same-cluster results for both models across all four workloads. The configurations and metrics follow Section 4.1.

Across all four workloads, AReaL-DTE consistently achieves the lowest latency and peak memory consumption. On Qwen3-30B-A3B, it is 6.8–7.6× faster than ByteCheckpoint-NCCL and 5.1–5.5× faster than PULSE-NCCL, while reducing peak GPU and CPU memory by 40.5–41.6% and 87.3–87.8%, respectively, relative to the lowest-memory baseline for each metric and workload. On

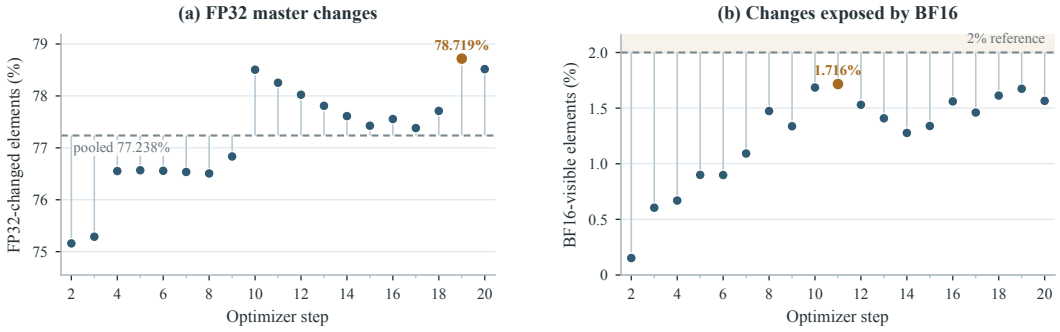


Figure 4: Per-step `FP32` master-weight and `BF16`-visible change rates over 19 effective optimizer updates (steps 2–20) in the Qwen3-0.6B training task. Step 1 is omitted because the learning-rate is set to zero due to warmup, yielding no parameter update. Each reported step contains 596,049,920 valid parameter elements. The dashed line in (a) denotes the average `FP32` change rate of 77.238%. In (b), all `BF16`-visible rates remain below 2%, with a maximum of 1.716% at step 11.

Qwen3-8B, the corresponding speedups are 3.9–4.0× and 4.6–7.4×, while the peak GPU and CPU memory reductions are 4.5–8.3% and 84.5–86.3%, respectively. The consistent latency and memory results on MATH-500 and LogiQA confirm that the improvements reported in the main text are not specific to the two representative workloads.

Table 6: Complete same-cluster results on **Qwen3-30B-A3B**.

Benchmark	Metric	AReaL-DTE	ByteCheckpoint		PULSE	
			NCCL	Disk	NCCL	Disk
GSM8K	Latency ↓	11.58 s	88.03 s	226.53 s	59.42 s	93.40 s
	GPU ↓	71.78 GB	121.82 GB	121.28 GB	129.17 GB	129.15 GB
	CPU ↓	15.33 GB	129.29 GB	125.53 GB	141.28 GB	139.19 GB
HumanEval	Latency ↓	11.83 s	83.43 s	227.21 s	64.80 s	101.16 s
	GPU ↓	71.45 GB	120.12 GB	121.20 GB	130.18 GB	129.20 GB
	CPU ↓	15.45 GB	126.65 GB	123.17 GB	139.68 GB	138.64 GB
MATH-500	Latency ↓	12.23 s	82.67 s	227.59 s	65.66 s	96.46 s
	GPU ↓	71.34 GB	122.03 GB	121.55 GB	129.01 GB	129.10 GB
	CPU ↓	15.44 GB	123.21 GB	122.79 GB	137.04 GB	139.83 GB
LogiQA	Latency ↓	12.14 s	89.48 s	228.27 s	62.00 s	92.14 s
	GPU ↓	71.19 GB	121.95 GB	121.87 GB	128.57 GB	128.29 GB
	CPU ↓	15.40 GB	123.80 GB	121.49 GB	141.60 GB	140.50 GB

Notes. Latency includes update construction, communication, and application. Memory is reported in decimal GB; lower values are better.

B.2 Complete Qwen3-8B Cross-Cluster Results

Table 8 reports the complete cross-cluster results for Qwen3-8B across all four benchmark workloads and all three effective bandwidths. The latency and memory metrics follow Section 4.1.

Across all workloads and bandwidths, AReaL-DTE achieves the lowest latency and peak memory consumption. Relative to the lowest-memory storage-mediated baseline for each metric and workload, AReaL-DTE reduces peak GPU and CPU memory by 13.7–14.2% and 16.0–25.5%, respectively. At 0.1 Gbps, it is 16.3–19.9× faster than ByteCheckpoint, while its advantage over the sparse PULSE baseline is smaller because network transfer dominates both methods. At 10 Gbps, the speedup over PULSE increases to 2.17–2.53× as local construction and application costs become more prominent. This trend is consistent with the Qwen3-30B-A3B results reported in the main text.

Table 7: Complete same-cluster results on **Qwen3-8B**.

Benchmark	Metric	AReaL-DTE	ByteCheckpoint		PULSE	
			NCCL	Disk	NCCL	Disk
GSM8K	Latency ↓	2.82 s	11.28 s	51.35 s	13.00 s	25.07 s
	GPU ↓	56.05 GB	61.91 GB	60.98 GB	67.97 GB	68.46 GB
	CPU ↓	3.45 GB	34.16 GB	22.21 GB	36.53 GB	36.96 GB
HumanEval	Latency ↓	2.86 s	11.26 s	50.14 s	15.94 s	27.21 s
	GPU ↓	57.92 GB	60.67 GB	60.75 GB	66.95 GB	68.02 GB
	CPU ↓	3.45 GB	34.08 GB	24.72 GB	36.53 GB	36.96 GB
MATH-500	Latency ↓	2.77 s	11.08 s	48.22 s	20.47 s	25.01 s
	GPU ↓	57.82 GB	61.89 GB	61.11 GB	66.86 GB	68.15 GB
	CPU ↓	3.48 GB	34.16 GB	25.27 GB	36.09 GB	36.63 GB
LogiQA	Latency ↓	2.87 s	11.28 s	48.70 s	15.44 s	26.51 s
	GPU ↓	56.06 GB	62.02 GB	61.14 GB	67.28 GB	68.51 GB
	CPU ↓	3.47 GB	34.13 GB	25.27 GB	36.43 GB	36.84 GB

Notes. Latency includes update construction, communication, and application. Memory is reported in decimal GB; lower values are better.

Table 8: Cross-cluster weight-synchronization results on **Qwen3-8B**. Lower values are better.

(a) Synchronization latency

Dataset	AReaL-DTE			ByteCheckpoint			PULSE		
	0.1 Gbps	1 Gbps	10 Gbps	0.1 Gbps	1 Gbps	10 Gbps	0.1 Gbps	1 Gbps	10 Gbps
GSM8K	117.40 s	20.47 s	10.78 s	2339.98 s	281.43 s	75.60 s	142.64 s	36.50 s	27.22 s
HumanEval	118.66 s	21.87 s	12.20 s	2341.15 s	280.49 s	74.41 s	127.20 s	38.35 s	29.28 s
MATH-500	127.59 s	22.94 s	12.48 s	2336.93 s	278.35 s	72.50 s	138.99 s	37.83 s	27.02 s
LogiQA	143.42 s	23.65 s	11.67 s	2340.49 s	279.15 s	73.00 s	158.78 s	41.02 s	28.80 s

(b) Peak memory consumption

Dataset	AReaL-DTE		ByteCheckpoint		PULSE	
	GPU	CPU	GPU	CPU	GPU	CPU
GSM8K	52.40 GB	18.65 GB	60.98 GB	22.21 GB	68.46 GB	36.96 GB
HumanEval	52.44 GB	20.37 GB	60.75 GB	24.72 GB	68.02 GB	36.96 GB
MATH-500	52.46 GB	18.82 GB	61.11 GB	25.27 GB	68.15 GB	36.63 GB
LogiQA	52.45 GB	20.48 GB	61.14 GB	25.27 GB	68.51 GB	36.84 GB

B.3 Sparsity and Compressed Payload Reduction

Figure 5 examines the source of the cross-cluster bandwidth savings. Across both models and all four workloads, fewer than 2% of the `BF16` weights change between consecutive policy versions. On Qwen3-30B-A3B, AReaL-DTE produces 2.53–2.87 GB compressed artifacts, compared with approximately 44.5 GB for ByteCheckpoint, yielding a $16.3\times$ average reduction. The average reduction on Qwen3-8B is $19.7\times$. The payload ratio is larger than the changed-weight fraction because each sparse entry stores both an index and its target `BF16` value.

B.4 Complete Ablation Results

We conduct an ablation study of AReaL-DTE on the storage-mediated path. We compare the complete design with three ablated variants on both Qwen3-8B and Qwen3-30B-A3B across four benchmarks.

"w/o inversion" removes the optimizer-inverted reconstruction described in Section 3.1; it instead retains a complete pre-update policy in host memory and stages the snapshot to the GPU for sparse change detection. "w/o inversion and remapping" additionally removes the pre-transfer receiver-space remapping described in Section 3.2. Consequently, each receiving rank must locally map and filter the canonical sparse changes before applying them. "w/o sparse transfer" uses the same storage-mediated path but synchronizes the complete policy instead of constructing a sparse payload.

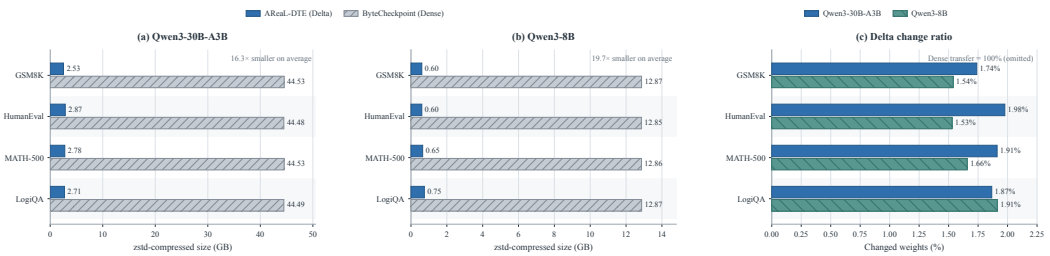


Figure 5: Zstd-compressed synchronization artifacts and changed-weight fraction. Panels (a) and (b) compare AReaL-DTE with ByteCheckpoint on Qwen3-30B-A3B and Qwen3-8B. Panel (c) reports the fraction of changed `BF16` weights; the 100% dense-transfer fraction is omitted.

AReaL-DTE denotes the complete design described in Section 3. All configurations use the same storage-mediated transport; components not explicitly ablated remain unchanged.

Table 9 shows that AReaL-DTE achieves the lowest latency and memory consumption for every model–benchmark combination. Relative to "w/o sparse transfer," AReaL-DTE is 6.5–$7.9\times$ faster on Qwen3-8B and 3.8–$4.5\times$ faster on Qwen3-30B-A3B. It additionally reduces peak GPU memory by 8.4–9.2% and 18.2–18.6%, respectively, and peak CPU memory by 44.2–49.8% and 21.2–28.9%. These results show that transmitting and applying a complete policy remains expensive even when both configurations use the same remapping implementation, storage-mediated transport path, and receiver-side application pipeline.

Retaining a historical snapshot enables sparse transfer but does not eliminate model-scale local state. Compared with "w/o inversion," AReaL-DTE is 1.91–$2.30\times$ faster on Qwen3-8B and 1.50–$1.60\times$ faster on Qwen3-30B-A3B. It reduces peak GPU memory by 51.3–51.5% and 45.7–46.3%, respectively, while reducing peak CPU memory by 66.7–69.9% and 46.8–51.8%. To accelerate change detection, "w/o inversion" stages the retained historical weights from the CPU to the GPU and compares them with the current weights there, which is faster than moving the current weights to the CPU for comparison. However, retaining the complete snapshot increases CPU memory, while staging it introduces additional transfer latency and causes both weight versions to coexist on the GPU, substantially increasing peak GPU memory.

Comparing "w/o inversion and remapping" with "w/o inversion" isolates the effect of pre-transfer receiver-space remapping. Removing this stage increases latency by 16.6–31.2% on Qwen3-8B and 20.3–23.6% on Qwen3-30B-A3B. Without receiver-ready delta entries, each inference rank must read the entire canonical sparse-delta payload, map its indices into the local layout, and filter out entries belonging to other shards before applying the remaining values. This receiver-side processing increases synchronization latency.

C Derivation of AdamW Inversion

This section derives the reconstruction rule in Equation 1. The derivation assumes element-wise arithmetic, the AdamW update used by our training stack, fixed moment coefficients, bias correction, multiplicative decoupled weight decay, and a second-moment denominator without an AMSGrad maximum.

Let $\boldsymbol{\theta}_{t-1}$ denote the `FP32` master-weight parameter before optimizer step t, let $\mathbf{g}_t$ denote its corresponding gradient, and let $\mathbf{m}_t$ and $\mathbf{v}_t$ denote the first- and second-moment states after incorporating $\mathbf{g}_t$. AdamW updates the moment states element-wise as

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t, \qquad \mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\left(\mathbf{g}_t \odot \mathbf{g}_t\right), \tag{27}$$

where all tensor operations in this derivation are element-wise. Under the standard initialization $\mathbf{m}_0 = \mathbf{0}$ and $\mathbf{v}_0 = \mathbf{0}$, the corresponding bias-corrected estimates are

Table 9: Ablation of disk-mediated policy synchronization on Qwen3-8B and Qwen3-30B-A3B across four benchmarks. Lower is better for all metrics.

Benchmark	Method	Qwen3-8B			Qwen3-30B-A3B		
		Latency (s) ↓	GPU peak (GB) ↓	CPU peak (GB) ↓	Latency (s) ↓	GPU peak (GB) ↓	CPU peak (GB) ↓
GSM8K	w/o inversion and remapping	23.21	109.39	61.91	48.32	133.00	76.75
	w/o inversion	19.91	108.02	62.01	40.05	133.16	77.39
	w/o sparse transfer	68.76	57.19	37.12	113.49	87.45	51.46
	AReaL-DTE	8.67	52.40	18.65	25.10	71.52	40.56
HumanEval	w/o inversion and remapping	26.44	108.52	63.10	50.07	133.89	78.83
	w/o inversion	20.20	107.62	63.12	41.45	133.88	77.88
	w/o sparse transfer	65.91	57.42	36.73	105.77	88.32	51.01
	AReaL-DTE	10.09	52.44	20.37	27.70	71.96	37.58
MATH-500	w/o inversion and remapping	25.06	109.68	62.63	51.04	133.29	76.76
	w/o inversion	19.61	108.17	62.35	41.28	132.35	76.25
	w/o sparse transfer	81.79	57.39	37.24	110.15	87.84	52.68
	AReaL-DTE	10.29	52.46	18.82	27.27	71.84	37.44
LogiQA	w/o inversion and remapping	25.04	109.43	62.61	49.57	133.47	76.78
	w/o inversion	19.09	107.97	61.59	41.20	133.61	76.58
	w/o sparse transfer	70.39	57.75	36.72	107.22	88.30	52.44
	AReaL-DTE	9.31	52.45	20.48	26.99	71.88	40.75

Notes. "w/o inversion" removes the optimizer-inverted reconstruction described in Section 3.1 and instead retains a pre-update snapshot. "w/o remapping" omits the pre-transfer receiver-space delta remapping described in Section 3.2; "w/o inversion and remapping" combines these two ablations. "w/o sparse transfer" synchronizes the complete policy instead of sparse `BF16` changes.

$$\widehat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t}, \qquad \widehat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t}. \tag{28}$$

Let η_t, λ, and ϵ denote the learning rate at optimizer step t, the weight-decay coefficient, and the numerical-stability constant for the parameter group containing $\boldsymbol{\theta}_{t-1}$, respectively. Under multiplicative decoupled weight decay, the AdamW parameter update is

$$\boldsymbol{\theta}_t = (1-\eta_t\lambda)\boldsymbol{\theta}_{t-1} - \eta_t \frac{\widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t}+\epsilon}. \tag{29}$$

Rearranging the forward update gives

$$(1-\eta_t\lambda)\boldsymbol{\theta}_{t-1} = \boldsymbol{\theta}_t + \eta_t \frac{\widehat{\mathbf{m}}_t}{\sqrt{\widehat{\mathbf{v}}_t}+\epsilon}. \tag{30}$$

Provided that $1-\eta_t\lambda \neq 0$, dividing by the weight-decay factor and substituting Equation 28 defines the reconstructed pre-update parameter as

$$\widehat{\boldsymbol{\theta}}_{t-1} = \frac{\boldsymbol{\theta}_t + \eta_t \dfrac{\mathbf{m}_t/(1-\beta_1^t)}{\sqrt{\mathbf{v}_t/(1-\beta_2^t)}+\epsilon}}{1-\eta_t\lambda}. \tag{31}$$

Here, $\widehat{\boldsymbol{\theta}}_{t-1}$ denotes the pre-update parameter reconstructed from the post-update optimizer state, whereas $\boldsymbol{\theta}_{t-1}$ denotes the actual parameter that existed immediately before optimizer step t.

Equation 31 provides the algebraic reconstruction used by AReaL-DTE. Because all quantities on its right-hand side remain available after optimizer step t, AReaL-DTE can compute $\widehat{\boldsymbol{\theta}}_{t-1}$ without retaining a historical copy of $\boldsymbol{\theta}_{t-1}$.

D MCORE AND HUGGING FACE PARAMETER LAYOUTS

This section illustrates why AREAL-DTE performs change detection in a canonical layout after training-to-inference conversion, rather than directly in the optimizer-native layout. We use a fused QKV weight matrix under grouped-query attention as an example; the same principle applies whenever the training and inference representations differ because of fusion, permutation, naming, or sharding.

D.1 OPTIMIZER-NATIVE MCORE LAYOUT

MCore organizes parameters for distributed training. Under tensor parallelism, each rank stores a local shard of the logical weight matrix, and multiple logical projections may be packed into one physical parameter. With a distributed optimizer, data-parallel ranks may additionally own disjoint slices of the `FP32` master weights and their optimizer states.

The master-weight slice owned by each rank is element-wise aligned with the corresponding slices of its first- and second-moment states. This alignment makes the optimizer-native layout the natural space in which to perform the reconstruction described in Section 3.1. However, a local MCore parameter block is a physical training representation and need not correspond one-to-one to a separately named inference parameter with the same element order or shard boundaries. Consequently, the values in such a block cannot be applied directly to an inference worker without accounting for differences in fusion, permutation, naming, and sharding between the two engines.

D.2 CANONICAL HUGGING FACE LAYOUT

The MCore-to-Hugging-Face converter defines a canonical parameter space shared by the reconstructed and current policy versions. Each logical inference weight appears as a separately named tensor with a fixed shape and element order.

For a grouped-query attention layer, the canonical projection tensors are

$$\mathbf{W}_Q \in \mathbb{R}^{(h_q d_h)\times d}, \qquad \mathbf{W}_K \in \mathbb{R}^{(h_{\mathrm{kv}} d_h)\times d}, \qquad \mathbf{W}_V \in \mathbb{R}^{(h_{\mathrm{kv}} d_h)\times d}, \tag{32}$$

where d is the input hidden dimension, d_h is the head dimension, h_q is the number of query heads, and h_{kv} is the number of key/value heads. These tensors correspond to the separately named `q_proj`, `k_proj`, and `v_proj` parameters exposed by the Hugging Face model interface.

The canonical representation does not need to match the physical sharding of an inference worker. Instead, it provides a common coordinate system in which the reconstructed and current versions have identical parameter names, shapes, and element orders. The transfer plan from Section 3.2 subsequently maps the canonical coordinates of changed elements into receiver-local coordinates.

D.3 EXAMPLE: FUSED QKV UNDER GROUPED-QUERY ATTENTION

Consider grouped-query attention with $h_{\mathrm{kv}} \mid h_q$. Each key/value head is shared by g query heads, where $g = \frac{h_q}{h_{\mathrm{kv}}}$ query heads. Let $\mathbf{Q}_i \in \mathbb{R}^{d_h \times d}$, for $i \in \{0, \dots, h_q - 1\}$, denote the row block associated with query head i. Similarly, let $\mathbf{K}_j, \mathbf{V}_j \in \mathbb{R}^{d_h \times d}$, for $j \in \{0, \dots, h_{\mathrm{kv}} - 1\}$, denote the row blocks associated with key head j and value head j, respectively.

In the MCore representation used by our training stack, each key/value head j forms a group with its g associated query heads. The query-head blocks in group j are $\mathbf{Q}_{jg+\ell}$, where $\ell \in \{0, \dots, g-1\}$. At the logical, unsharded level, MCore concatenates these groups along the output-row dimension as

$$\mathbf{W}_{\mathrm{QKV}}^{\mathrm{MCore}} = \mathrm{concat}_{j=0}^{h_{\mathrm{kv}}-1} \left[\mathbf{Q}_{jg}, \mathbf{Q}_{jg+1}, \dots, \mathbf{Q}_{jg+(g-1)}, \mathbf{K}_j, \mathbf{V}_j\right]. \tag{33}$$

Thus, each group contains its g query-head blocks followed by the corresponding key-head and value-head blocks, and the groups are ordered by increasing j. Its logical, unsharded shape is therefore

$$\mathbf{W}_{\mathrm{QKV}}^{\mathrm{MCore}} \in \mathbb{R}^{((h_q+2h_{\mathrm{kv}})d_h)\times d}. \tag{34}$$

Let P_{tp} denote the tensor-parallel degree. For each tensor-parallel rank $r_{\mathrm{tp}} \in \{0,\dots,P_{\mathrm{tp}}-1\}$, let $\mathcal{R}_{r_{\mathrm{tp}}}$ denote the set of output-row indices assigned to that rank by the MCore QKV sharding rule. The corresponding rank-local tensor is

$$\mathbf{W}_{\mathrm{QKV},r_{\mathrm{tp}}}^{\mathrm{MCore}} = \mathbf{W}_{\mathrm{QKV}}^{\mathrm{MCore}}[\mathcal{R}_{r_{\mathrm{tp}}},:]. \tag{35}$$

When a distributed optimizer further partitions the `FP32` master weight, each owning rank first reconstructs its element-wise aligned local slice using Equation 31. The optimizer group then gathers the slices required to form the corresponding tensor-parallel parameter for conversion.

The Hugging Face representation instead stores the query, key, and value projections as three separately named logical parameters. Concatenating their head-specific row blocks gives

$$\mathbf{W}_Q = \mathrm{concat}_{i=0}^{h_q-1}\left[\mathbf{Q}_i\right], \qquad \mathbf{W}_K = \mathrm{concat}_{j=0}^{h_{\mathrm{kv}}-1}\left[\mathbf{K}_j\right], \qquad \mathbf{W}_V = \mathrm{concat}_{j=0}^{h_{\mathrm{kv}}-1}\left[\mathbf{V}_j\right]. \tag{36}$$

Each concatenation is performed along the output-row dimension in increasing head-index order. The query-head blocks that form $\mathbf{W}_Q$ need not be contiguous in $\mathbf{W}_{\mathrm{QKV}}^{\mathrm{MCore}}$, because each group of query-head blocks is followed by its corresponding key-head and value-head blocks. Thus, the parameter boundaries and element order in the fused MCore representation do not match those of the separately named Hugging Face parameters. AReaL-DTE therefore applies the same MCore-to-Hugging-Face converter to the reconstructed and current tensors before change detection. This conversion produces aligned canonical tensors with identical parameter names, shapes, and element orders, enabling element-wise comparison in the inference-visible `BF16` parameter space.

Given a fused MCore tensor $\mathbf{W}^{\mathrm{MCore}}$, the MCore-to-Hugging-Face converter gathers the required tensor-parallel pieces, unpacks the grouped QKV blocks, and emits three separately named canonical parameters:

$$\mathrm{convert}_{\mathrm{m2h}}\left(\mathbf{W}^{\mathrm{MCore}}\right) = \{\texttt{q_proj} \mapsto \mathbf{W}_Q, \texttt{k_proj} \mapsto \mathbf{W}_K, \texttt{v_proj} \mapsto \mathbf{W}_V\}. \tag{37}$$

The converter is applied separately to the reconstructed pre-update MCore tensor and the current MCore tensor. Because both versions undergo the same gathering, unpacking, permutation, and naming operations, their canonical outputs have identical parameter names, shapes, and element orders. This alignment enables the inference-visible `BF16` comparison in Equation 3.

The resulting sparse changes are expressed using canonical parameter names and coordinates rather than coordinates in the packed MCore representation. Subsequently, the remapping in Equation 4 translates these canonical coordinates into receiver-local coordinates.

E Storage-Mediated Cross-Cluster Transfer

Within a cluster, training and inference workers can join a common GPU communication group and exchange receiver-ready payloads directly between GPU processes. Across clusters, however, these workers reside in separate network and resource-management domains and cannot form the same point-to-point communication group. AReaL-DTE therefore replaces direct GPU data movement with an HTTP-backed shared filesystem while preserving the receiver-ready sparse representation produced by Section 3.2.

On the training side, AReaL-DTE stages the converted tensors through a bounded pinned-memory pipeline and serializes only the changed index–value entries into compressed chunks. Writer ranks publish their assigned chunks, and a designated coordinator commits a version manifest only after publication completes. The manifest records the version chain, payload locations, lengths, and checksums, allowing the receiving service to discover and validate a complete update. If a policy version contains no changed payload, AReaL-DTE retains the previous synchronization base instead of publishing an empty version.

After retrieving a committed manifest, the receiving inference service first verifies that the update extends its locally applied base version. It then reads only the sparse chunks belonging to that version. Payload files are processed as streams rather than loaded as an entire version, while multiple worker threads concurrently perform file reading, checksum verification, and Zstandard decompression. The resulting sparse entries are passed through the weight-loading path in batches of at most 512 MiB to bound transient host-memory usage. During application, a scoped copy mechanism overwrites only the changed positions while preserving all unchanged GPU-resident weights. Consequently, the receiver neither constructs a complete local Hugging Face checkpoint nor performs a full-model reload from storage.